UNITED STATES PATENT APPLICATION

of

Alexander T. Garthwaite

for a

CONCURRENT NON-INTRUSIVE PROCESSING OF A CARD TABLE

SUMMARIZING MODIFIED REFERENCE LOCATIONS

CONCURRENT NON-INTRUSIVE PROCESSING OF A CARD TABLE SUMMARIZING MODIFIED REFERENCE LOCATIONS

BACKGROUND OF THE INVENTION

Field of the Invention

5   The present invention is directed to memory management. It particularly concerns what has come to be known as "garbage collection."

Background Information

In the field of computer systems, considerable effort has been expended on the task of allocating memory to data objects. For the purposes of this discussion, the term

10   *object* refers to a data structure represented in a computer system's memory. Other terms sometimes used for the same concept are *record* and *structure*. An object may be identified by a *reference*, a relatively small amount of information that can be used to access the object. A reference can be represented as a "pointer" or a "machine address," which may require, for instance, only sixteen, thirty-two, or sixty-four bits of information, al-

15   though there are other ways to represent a reference.

In some systems, which are usually known as "object oriented," objects may have associated *methods*, which are routines that can be invoked by reference to the object. They also may belong to a *class*, which is an organizational entity that may contain method code or other information shared by all objects belonging to that class. In the

20   discussion that follows, though, the term *object* will not be limited to such structures; it will additionally include structures with which methods and classes are not associated.

The invention to be described below is applicable to systems that allocate memory to objects dynamically. Not all systems employ dynamic allocation. In some computer

1

languages, source programs must be so written that all objects to which the program's variables refer are bound to storage locations at compile time. This storage-allocation approach, sometimes referred to as "static allocation," is the policy traditionally used by the Fortran programming language, for example.

5   Even for compilers that are thought of as allocating objects only statically, of course, there is often a certain level of abstraction to this binding of objects to storage locations. Consider the typical computer system 10 depicted in Fig. 1, for example. Data, and instructions for operating on them, that a microprocessor 11 uses may reside in on-board cache memory or be received from further cache memory 12, possibly through 10   the mediation of a cache controller 13. That controller 13 can in turn receive such data from system read/write memory ("RAM") 14 through a RAM controller 15 or from various peripheral devices through a system bus 16. The memory space made available to an application program may be "virtual" in the sense that it may actually be considerably larger than RAM 14 provides. So the RAM contents will be swapped to and from a sys-15   tem disk 17.

Additionally, the actual physical operations performed to access some of the most-recently visited parts of the process's address space often will actually be performed in the cache 12 or in a cache on board microprocessor 11 rather than on the RAM 14, with which those caches swap data and instructions just as RAM 14 and system disk 17 20   do with each other.

A further level of abstraction results from the fact that an application will often be run as one of many processes operating concurrently with the support of an underlying operating system. As part of that system's memory management, the application's memory space may be moved among different actual physical locations many times in order to 25   allow different processes to employ shared physical memory devices. That is, the location specified in the application's machine code may actually result in different physical locations at different times because the operating system adds different offsets to the machine-language-specified location.

Despite these expedients, the use of static memory allocation in writing certain 30   long-lived applications makes it difficult to restrict storage requirements to the available

memory space. Abiding by space limitations is easier when the platform provides for dynamic memory allocation, i.e., when memory space to be allocated to a given object is determined only at run time.

Dynamic allocation has a number of advantages, among which is that the run-time
5   system is able to adapt allocation to run-time conditions. For example, the programmer can specify that space should be allocated for a given object only in response to a particular run-time condition. The C-language library function malloc() is often used for this purpose. Conversely, the programmer can specify conditions under which memory previously allocated to a given object can be reclaimed for reuse. The C-language library
10   function free() results in such memory reclamation.

Because dynamic allocation provides for memory reuse, it facilitates generation of large or long-lived applications, which over the course of their lifetimes may employ objects whose total memory requirements would greatly exceed the available memory resources if they were bound to memory locations statically.

15   Particularly for long-lived applications, though, allocation and reclamation of dynamic memory must be performed carefully. If the application fails to reclaim unused memory—or, worse, loses track of the address of a dynamically allocated segment of memory—its memory requirements will grow over time to exceed the system's available memory. This kind of error is known as a "memory leak."

20   Another kind of error occurs when an application reclaims memory for reuse even though it still maintains a reference to that memory. If the reclaimed memory is reallocated for a different purpose, the application may inadvertently manipulate the same memory in multiple inconsistent ways. This kind of error is known as a "dangling reference," because an application should not retain a reference to a memory location once
25   that location is reclaimed. Explicit dynamic-memory management by using interfaces like malloc()/free() often leads to these problems.

A way of reducing the likelihood of such leaks and related errors is to provide memory-space reclamation in a more-automatic manner. Techniques used by systems that reclaim memory space automatically are commonly referred to as "garbage collec-

tion." Garbage collectors operate by reclaiming space that they no longer consider "reachable." Statically allocated objects represented by a program's global variables are normally considered reachable throughout a program's life. Such objects are not ordinarily stored in the garbage collector's managed memory space, but they may contain references to dynamically allocated objects that are, and such objects are considered reachable. Clearly, an object referred to in the processor's call stack is reachable, as is an object referred to by register contents. And an object referred to by any reachable object is also reachable.

The use of garbage collectors is advantageous because, whereas a programmer working on a particular sequence of code can perform his task creditably in most respects with only local knowledge of the application at any given time, memory allocation and reclamation require a global knowledge of the program. Specifically, a programmer dealing with a given sequence of code does tend to know whether some portion of memory is still in use for that sequence of code, but it is considerably more difficult for him to know what the rest of the application is doing with that memory. By tracing references from some conservative notion of a "root set," e.g., global variables, registers, and the call stack, automatic garbage collectors obtain global knowledge in a methodical way. By using a garbage collector, the programmer is relieved of the need to worry about the application's global state and can concentrate on local-state issues, which are more manageable. The result is applications that are more robust, having no dangling references and fewer memory leaks.

Garbage-collection mechanisms can be implemented by various parts and levels of a computing system. One approach is simply to provide them as part of a batch compiler's output. Consider Fig. 2's simple batch-compiler operation, for example. A computer system executes in accordance with compiler object code and therefore acts as a compiler 20. The compiler object code is typically stored on a medium such as Fig. 1's system disk 17 or some other machine-readable medium, and it is loaded into RAM 14 to configure the computer system to act as a compiler. In some cases, though, the compiler object code's persistent storage may instead be provided in a server system remote from the machine that performs the compiling. The electrical signals that carry the digital data

4

by which the computer systems exchange that code are examples of the kinds of electro-magnetic signals by which the computer instructions can be communicated. Others are radio waves, microwaves, and both visible and invisible light.

The input to the compiler is the application source code, and the end product of the compiler process is application object code. This object code defines an application 21, which typically operates on input such as mouse clicks, etc., to generate a display or some other type of output. This object code implements the relationship that the programmer intends to specify by his application source code. In one approach to garbage collection, the compiler 20, without the programmer's explicit direction, additionally generates code that automatically reclaims unreachable memory space.

Even in this simple case, though, there is a sense in which the application does not itself provide the entire garbage collector. Specifically, the application will typically call upon the underlying operating system's memory-allocation functions. And the operating system may in turn take advantage of various hardware that lends itself particularly to use in garbage collection. So even a very simple system may disperse the garbage-collection mechanism over a number of computer-system layers.

To get some sense of the variety of system components that can be used to implement garbage collection, consider Fig. 3's example of a more complex way in which various levels of source code can result in the machine instructions that a processor executes. In the Fig. 3 arrangement, the human applications programmer produces source code 22 written in a high-level language. A compiler 23 typically converts that code into "class files." These files include routines written in instructions, called "byte codes" 24, for a "virtual machine" that various processors can be software-configured to emulate. This conversion into byte codes is almost always separated in time from those codes' execution, so Fig. 3 divides the sequence into a "compile-time environment" 25 separate from a "run-time environment" 26, in which execution occurs. One example of a high-level language for which compilers are available to produce such virtual-machine instructions is the Java™ programming language. (*Java* is a trademark or registered trademark of Sun Microsystems, Inc., in the United States and other countries.)

Most typically, the class files' byte-code routines are executed by a processor under control of a virtual-machine process 27. That process emulates a virtual machine from whose instruction set the byte codes are drawn. As is true of the compiler 23, the virtual-machine process 27 may be specified by code stored on a local disk or some other

5 machine-readable medium from which it is read into Fig. 1's RAM 14 to configure the computer system to implement the garbage collector and otherwise act as a virtual machine. Again, though, that code's persistent storage may instead be provided by a server system remote from the processor that implements the virtual machine, in which case the code would be transmitted electrically or optically to the virtual-machine-implementing

10 processor.

In some implementations, much of the virtual machine's action in executing these byte codes is most like what those skilled in the art refer to as "interpreting," so Fig. 3 depicts the virtual machine as including an "interpreter" 28 for that purpose. In addition to or instead of running an interpreter, many virtual-machine implementations actually

15 compile the byte codes concurrently with the resultant object code's execution, so Fig. 3 depicts the virtual machine as additionally including a "just-in-time" compiler 29. We will refer to the just-in-time compiler and the interpreter together as "execution engines" since they are the methods by which byte code can be executed.

Now, some of the functionality that source-language constructs specify can be

20 quite complicated, requiring many machine-language instructions for their implementation. One quite-common example is a source-language instruction that calls for 64-bit arithmetic on a 32-bit machine. More germane to the present invention is the operation of dynamically allocating space to a new object; the allocation of such objects must be mediated by the garbage collector.

25 In such situations, the compiler may produce "inline" code to accomplish these operations. That is, all object-code instructions for carrying out a given source-code-prescribed operation will be repeated each time the source code calls for the operation. But inlining runs the risk that "code bloat" will result if the operation is invoked at many source-code locations.

The natural way of avoiding this result is instead to provide the operation's implementation as a procedure, i.e., a single code sequence that can be called from any location in the program. In the case of compilers, a collection of procedures for implementing many types of source-code-specified operations is called a *runtime system* for the language.

5　The execution engines and the runtime system of a virtual machine are designed together so that the engines "know" what runtime-system procedures are available in the virtual machine (and on the target system if that system provides facilities that are directly usable by an executing virtual-machine program.) So, for example, the just-in-time compiler 29 may generate native code that includes calls to memory-allocation procedures

10　provided by the virtual machine's runtime system. These allocation routines may in turn invoke garbage-collection routines of the runtime system when there is not enough memory available to satisfy an allocation. To represent this fact, Fig. 3 includes block 30 to show that the compiler's output makes calls to the runtime system as well as to the operating system 31, which consists of procedures that are similarly system-resident but are

15　not compiler-dependent.

Although the Fig. 3 arrangement is a popular one, it is by no means universal, and many further implementation types can be expected. Proposals have even been made to implement the virtual machine 27's behavior in a hardware processor, in which case the hardware itself would provide some or all of the garbage-collection function.

20　The arrangement of Fig. 3 differs from Fig. 2 in that the compiler 23 for converting the human programmer's code does not contribute to providing the garbage-collection function; that results largely from the virtual machine 27's operation. Those skilled in that art will recognize that both of these organizations are merely exemplary, and many modern systems employ hybrid mechanisms, which partake of the characteristics

25　tics of traditional compilers and traditional interpreters both.

The invention to be described below is applicable independently of whether a batch compiler, a just-in-time compiler, an interpreter, or some hybrid is employed to process source code. In the remainder of this application, therefore, we will use the term *compiler* to refer to any such mechanism, even if it is what would more typically be

30　called an interpreter.

In short, garbage collectors can be implemented in a wide range of combinations of hardware and/or software. As is true of most of the garbage-collection techniques described in the literature, the invention to be described below is applicable to most such systems.

5   By implementing garbage collection, a computer system can greatly reduce the occurrence of memory leaks and other software deficiencies in which human programming frequently results. But it can also have significant adverse performance effects if it is not implemented carefully. To distinguish the part of the program that does "useful" work from that which does the garbage collection, the term *mutator* is sometimes used in
10   discussions of these effects; from the collector's point of view, what the mutator does is mutate active data structures' connectivity.

Some garbage-collection approaches rely heavily on interleaving garbage-collection steps among mutator steps. In one type of garbage-collection approach, for instance, the mutator operation of writing a reference is followed immediately by gar-
15   bage-collector steps used to maintain a reference count in that object's header, and code for subsequent new-object storage includes steps for finding space occupied by objects whose reference count has fallen to zero. Obviously, such an approach can slow mutator operation significantly.

Other approaches therefore interleave very few garbage-collector-related instruc-
20   tions into the main mutator process but instead interrupt it from time to time to perform garbage-collection cycles, in which the garbage collector finds unreachable objects and reclaims their memory space for reuse. Such an approach will be assumed in discussing Fig. 4's depiction of a simple garbage-collection operation. Within the memory space allocated to a given application is a part 40 managed by automatic garbage collection. In
25   the following discussion, this will be referred to as the "heap," although in other contexts that term refers to all dynamically allocated memory. During the course of the application's execution, space is allocated for various objects 42, 44, 46, 48, and 50. Typically, the mutator allocates space within the heap by invoking the garbage collector, which at some level manages access to the heap. Basically, the mutator asks the garbage collector
30   for a pointer to a heap region where it can safely place the object's data. The garbage

collector keeps track of the fact that the thus-allocated region is occupied. It will refrain from allocating that region in response to any other request until it determines that the mutator no longer needs the region allocated to that object.

Garbage collectors vary as to which objects they consider reachable and unreachable. For the present discussion, though, an object will be considered "reachable" if it is referred to, as object 42 is, by a reference in the root set 52. The root set consists of reference values stored in the mutator's threads' call stacks, the CPU registers, and global variables outside the garbage-collected heap. An object is also reachable if it is referred to, as object 46 is, by another reachable object (in this case, object 42). Objects that are not reachable can no longer affect the program, so it is safe to re-allocate the memory spaces that they occupy.

A typical approach to garbage collection is therefore to identify all reachable objects and reclaim any previously allocated memory that the reachable objects do not occupy. A typical garbage collector may identify reachable objects by tracing references from the root set 52. For the sake of simplicity, Fig. 4 depicts only one reference from the root set 52 into the heap 40. (Those skilled in the art will recognize that there are many ways to identify references, or at least data contents that may be references.) The collector notes that the root set points to object 42, which is therefore reachable, and that reachable object 42 points to object 46, which therefore is also reachable. But those reachable objects point to no other objects, so objects 44, 48, and 50 are all unreachable, and their memory space may be reclaimed. This may involve, say, placing that memory space in a list of free memory blocks.

To avoid excessive heap fragmentation, some garbage collectors additionally relocate reachable objects. Fig. 5 shows a typical approach. The heap is partitioned into two halves, hereafter called "semi-spaces." For one garbage-collection cycle, all objects are allocated in one semi-space 54, leaving the other semi-space 56 free. When the garbage-collection cycle occurs, objects identified as reachable are "evacuated" to the other semi-space 56, so all of semi-space 54 is then considered free. Once the garbage-collection cycle has occurred, all new objects are allocated in the lower semi-space 56

until yet another garbage-collection cycle occurs, at which time the reachable objects are evacuated back to the upper semi-space 54.

Although this relocation requires the extra steps of copying the reachable objects and updating references to them, it tends to be quite efficient, since most new objects quickly become unreachable, so most of the current semi-space is actually garbage. That is, only a relatively few, reachable objects need to be relocated, after which the entire semi-space contains only garbage and can be pronounced free for reallocation.

Now, a collection cycle can involve following all reference chains from the basic root set—i.e., from inherently reachable locations such as the call stacks, class statics and other global variables, and registers—and reclaiming all space occupied by objects not encountered in the process. And the simplest way of performing such a cycle is to interrupt the mutator to provide a collector interval in which the entire cycle is performed before the mutator resumes. For certain types of applications, this approach to collection-cycle scheduling is acceptable and, in fact, highly efficient.

For many interactive and real-time applications, though, this approach is not acceptable. The delay in mutator operation that the collection cycle's execution causes can be annoying to a user and can prevent a real-time application from responding to its environment with the required speed. In some applications, choosing collection times opportunistically can reduce this effect. Collection intervals can be inserted when an interactive mutator reaches a point at which it awaits user input, for instance.

So it may often be true that the garbage-collection operation's effect on performance can depend less on the total collection time than on when collections actually occur. But another factor that often is even more determinative is the duration of any single collection interval, i.e., how long the mutator must remain quiescent at any one time. In an interactive system, for instance, a user may never notice hundred-millisecond interruptions for garbage collection, whereas most users would find interruptions lasting for two seconds to be annoying.

The cycle may therefore be divided up among a plurality of collector intervals. When a collection cycle is divided up among a plurality of collection intervals, it is only

after a number of intervals that the collector will have followed all reference chains and be able to identify as garbage any objects not thereby reached. This approach is more complex than completing the cycle in a single collection interval; the mutator will usually modify references between collection intervals, so the collector must repeatedly update its view of the reference graph in the midst of the collection cycle. To make such updates practical, the mutator must communicate with the collector to let it know what reference changes are made between intervals.

An even more complex approach, which some systems use to eliminate discrete pauses or maximize resource-use efficiency, is to execute the mutator and collector in concurrent execution threads. Most systems that use this approach use it for most but not all of the collection cycle; the mutator is usually interrupted for a short collector interval, in which a part of the collector cycle takes place without mutation.

Independent of whether the collection cycle is performed concurrently with mutator operation, is completed in a single interval, or extends over multiple intervals is the question of whether the cycle is complete, as has tacitly been assumed so far, or is instead "incremental." In *incremental* collection, a collection cycle constitutes only an increment of collection: the collector does not follow all reference chains from the basic root set completely. Instead, it concentrates on only a portion, or *collection set*, of the heap. Specifically, it identifies every collection-set object referred to by a reference chain that extends into the collection set from outside of it, and it reclaims the collection-set space not occupied by such objects, possibly after evacuating them from the collection set.

By thus culling objects referenced by reference chains that do not necessarily originate in the basic root set, the collector can be thought of as expanding the root set to include as roots some locations that may not be reachable. Although incremental collection thereby leaves "floating garbage," it can result in relatively low pause times even if entire collection increments are completed during respective single collection intervals.

Most collectors that employ incremental collection operate in "generations," although this is not necessary in principle. Different portions, or *generations*, of the heap are subject to different collection policies. New objects are allocated in a "young" generation, and older objects are promoted from younger generations to older or more "ma-

11

ture" generations. Collecting the younger generations more frequently than the others yields greater efficiency because the younger generations tend to accumulate garbage faster; newly allocated objects tend to "die," while older objects tend to "survive."

But generational collection greatly increases what is effectively the root set for a given generation. Consider Fig. 6, which depicts a heap as organized into three generations 58, 60, and 62. Assume that generation 60 is to be collected. The process for this individual generation may be more or less the same as that described in connection with Figs. 4 and 5 for the entire heap, with one major exception. In the case of a single generation, the root set must be considered to include not only the call stack, registers, and global variables represented by set 52 but also objects in the other generations 58 and 62, which themselves may contain references to objects in generation 60. So pointers must be traced not only from the basic root set 52 but also from objects within the other generations.

One could perform this tracing by simply inspecting all references in all other generations at the beginning of every collection interval, and it turns out that this approach is actually feasible in some situations. But it takes too long in other situations, so workers in this field have employed a number of approaches to expediting reference tracing. One approach is to include so-called write barriers in the mutator process. A *write barrier* is code added to a write operation to record information from which the collector can determine where references were written or may have been since the last collection interval. A reference list can then be maintained by taking such a list as it existed at the end of the previous collection interval and updating it by inspecting only locations identified by the write barrier as possibly modified since the last collection interval.

One of the many write-barrier implementations commonly used by workers in this art employs what has been referred to as the "card table." Fig. 6 depicts the various generations as being divided into smaller sections, known for this purpose as "cards." Card tables 64, 66, and 68 associated with respective generations contain an entry for each of their cards. When the mutator writes a reference in a card, it makes an appropriate entry in the card-table location associated with that card (or, say, with the card in which the

object containing the reference begins). Most write-barrier implementations simply make a Boolean entry indicating that the write operation has been performed, although some may be more elaborate. The mutator having thus left a record of where new or modified references may be, the collector can thereafter prepare appropriate summaries of that in-

5 formation, as will be explained in due course. For the sake of concreteness, we will assume that the summaries are maintained by steps that occur principally at the beginning of each collection interval.

Of course, there are other write-barrier approaches, such as simply having the write barrier add to a list of addresses where references where written. Also, although

10 there is no reason in principle to favor any particular number of generations, and although Fig. 6 shows three, most generational garbage collectors have only two generations, of which one is the young generation and the other is the mature generation. Moreover, although Fig. 6 shows the generations as being of the same size, a more-typical configuration is for the young generation to be considerably smaller. Finally, although we as-

15 sumed for the sake of simplicity that collection during a given interval was limited to only one generation, a more-typical approach is actually to collect the whole young generation at every interval but to collect the mature one less frequently.

Some collectors collect the entire young generation in every interval and may thereafter perform mature-generation collection in the same interval. It may therefore

20 take relatively little time to scan all young-generation objects remaining after young-generation collection to find references into the mature generation. Even when such collectors do use card tables, therefore, they often do not use them for finding young-generation references that refer to mature-generation objects. On the other hand, laboriously scanning the entire mature generation for references to young-generation (or ma-

25 ture-generation) objects would ordinarily take too long, so the collector uses the card table to limit the amount of memory it searches for mature-generation references.

Now, although it typically takes very little time to collect the young generation, it may take more time than is acceptable within a single garbage-collection cycle to collect the entire mature generation. So some garbage collectors may collect the mature genera-

30 tion incrementally; that is, they may perform only a part of the mature generation's col-

lection during any particular collection cycle. Incremental collection presents the problem that, since the generation's unreachable objects outside the "collection set" of objects processed during that cycle cannot be recognized as unreachable, collection-set objects to which they refer tend not to be, either.

5   To reduce the adverse effect this would otherwise have on collection efficiency, workers in this field have employed the "train algorithm," which Fig. 7 depicts. A generation to be collected incrementally is divided into sections, which for reasons about to be described are referred to as "car sections." Conventionally, a generation's incremental collection occurs in fixed-size sections, and a car section's size is that of the generation

10   portion to be collected during one cycle.

The discussion that follows will occasionally employ the nomenclature in the literature by using the term *car* instead of *car section*. But the literature seems to use that term to refer variously not only to memory sections themselves but also to data structures that the train algorithm employs to manage them when they contain objects, as well as to

15   the more-abstract concept that the car section and managing data structure represent in discussions of the algorithm. So the following discussion will more frequently use the expression *car section* to emphasize the actual sections of memory space for whose management the car concept is employed.

According to the train algorithm, the car sections are grouped into "trains," which

20   are ordered, conventionally according to age. For example, Fig. 7 shows an oldest train 73 consisting of a generation 74's three car sections described by associated data structures 75, 76, and 78, while a second train 80 consists only of a single car section, represented by structure 82, and the youngest train 84 (referred to as the "allocation train") consists of car sections that data structures 86 and 88 represent. As will be seen

25   below, car sections' train memberships can change, and any car section added to a train is typically added to the end of a train.

Conventionally, the car collected in an increment is the one added earliest to the oldest train, which in this case is car 75. All of the generation's cars can thus be thought of as waiting for collection in a single long line, in which cars are ordered in accordance

with the order of the trains to which they belong and, within trains, in accordance with the order in which they were added to those trains.

As is usual, the way in which reachable objects are identified is to determine whether there are references to them in the root set or in any other object already determined to be reachable. In accordance with the train algorithm, the collector additionally performs a test to determine whether there are any references at all from outside the oldest train to objects within it. If there are not, then all cars within the train can be reclaimed, even though not all of those cars are in the collection set. And the train algorithm so operates that inter-car references tend to be grouped into trains, as will now be explained.

To identify references into the car from outside of it, train-algorithm implementations typically employ "remembered sets." As card tables are, remembered sets are used to keep track of references. Whereas a card-table entry contains information about references that the associated card contains, though, a remembered set associated with a given region contains information about references into that region from locations outside of it. In the case of the train algorithm, remembered sets are associated with car sections. Each remembered set, such as car 75's remembered set 90, lists locations in the generation that contain references into the associated car section.

The remembered sets for all of a generation's cars are typically updated at the start of each collection cycle. To illustrate how such updating and other collection operations may be carried out, Figs. 8A and 8B (together, "Fig. 8") depict an operational sequence in a system of the typical type mention above. That is, it shows a sequence of operations that may occur in a system in which the entire garbage-collected heap is divided into two generations, namely, a young generation and an old generation, and in which the young generation is much smaller than the old generation. Fig. 8 is also based on the assumption and that the train algorithm is used only for collecting the old generation.

Block 102 represents a period of the mutator's operation. As was explained above, the mutator makes a card-table entry to identify any card that it has "dirtied" by adding or modifying a reference that the card contains. At some point, the mutator will

be interrupted for collector operation. Different implementations employ different events to trigger such an interruption, but we will assume for the sake of concreteness that the system's dynamic-allocation routine causes such interruptions when no room is left in the young generation for any further allocation. A dashed line 103 represents the transition from mutator operation and collector operation.

In the system assumed for the Fig. 8 example, the collector collects the (entire) young generation each time such an interruption occurs. When the young generation's collection ends, the mutator operation usually resumes, without the collector's having collected any part of the old generation. Once in a while, though, the collector also collects part of the old generation, and Fig. 8 is intended to illustrate such an occasion.

When the collector's interval first starts, it first processes the card table, in an operation that block 104 represents. As was mentioned above, the collector scans the "dirtied" cards for references into the young generation. If a reference is found, that fact is memorialized appropriately. If the reference refers to a young-generation object, for example, an expanded card table may be used for this purpose. For each card, such an expanded card table might include a multi-byte array used to summarize the card's reference contents. The summary may, for instance, be a list of offsets that indicate the exact locations within the card of references to young-generation objects, or it may be a list of fine-granularity "sub-cards" within which references to young-generation objects may be found. If the reference refers to an old-generation object, the collector often adds an entry to the remembered set associated with the car containing that old-generation object. The entry identifies the reference's location, or at least a small region in which the reference can be found. For reasons that will become apparent, though, the collector will typically not bother to place in the remembered set the locations of references from objects in car sections farther forward in the collection queue than the referred-to object, i.e., from objects in older trains or in cars added earlier to the same train.

The collector then collects the young generation, as block 105 indicates. (Actually, young-generation collection may be interleaved with the dirty-region scanning, but the drawing illustrates it for purpose of explanation as being separate.) If a young-generation object is referred to by a reference that card-table scanning has revealed, that

16

object is considered to be potentially reachable, as is any young-generation object referred to by a reference in the root set or in another reachable young-generation object. The space occupied by any young-generation object thus considered reachable is withheld from reclamation. For example, it may be evacuated to a young-generation semi-

5   space that will be used for allocation during the next mutator interval. It may instead be promoted into the older generation, where it is placed into a car containing a reference to it or into a car in the last train. Or some other technique may be used to keep the memory space it occupies off the system's free list. The collector then reclaims any young-generation space occupied by any other objects, i.e., by any young-generation objects not

10   identified as transitively reachable through references located outside the young generation.

The collector then performs the train algorithm's central test, referred to above, of determining whether there are any references into the oldest train from outside of it. As was mentioned above, the actual process of determining, for each object, whether it can

15   be identified as unreachable is performed for only a single car section in any cycle. In the absence of features such as those provided by the train algorithm, this would present a problem, because garbage structures may be larger than a car section. Objects in such structures would therefore (erroneously) appear reachable, since they are referred to from outside the car section under consideration. But the train algorithm additionally keeps

20   track of whether there are any references into a given car from outside the train to which it belongs, and trains' sizes are not limited. As will be apparent presently, objects not found to be unreachable are relocated in such a way that garbage structures tend to be gathered into respective trains into which, eventually, no references from outside the train point. If no references from outside the train point to any objects inside the train, the

25   train can be recognized as containing only garbage. This is the test that block 106 represents. All cars in a train thus identified as containing only garbage can be reclaimed.

The question of whether old-generation references point into the train from outside of it is (conservatively) answered in the course of updating remembered sets; in the course of updating a car's remembered set, it is a simple matter to flag the car as being

30   referred to from outside the train. The step-106 test additionally involves determining

whether any references from outside the old generation point into the oldest train. Various approaches to making this determination have been suggested, including the conceptually simple approach of merely following all reference chains from the root set until those chains (1) terminate, (2) reach an old-generation object outside the oldest train, or

5   (3) reach an object in the oldest train. In the two-generation example, most of this work can be done readily by identifying references into the collection set from live young-generation objects during the young-generation collection. If one or more such chains reach the oldest train, that train includes reachable objects. It may also include reachable objects if the remembered-set-update operation has found one or more references into the

10 oldest train from outside of it. Otherwise, that train contains only garbage, and the collector reclaims all of its car sections for reuse, as block 107 indicates. The collector may then return control to the mutator, which resumes execution, as Fig. 8B's block 108 indicates.

If the train contains reachable objects, on the other hand, the collector turns to

15 evacuating potentially reachable objects from the collection set. The first operation, which block 110 represents, is to remove from the collection set any object that is reachable from the root set by way of a reference chain that does not pass through the part of the old generation that is outside of the collection set. In the illustrated arrangement, in which there are only two generations, and the young generation has previously been

20 completely collected during the same interval, this means evacuating from a collection set any object that (1) is directly referred to by a reference in the root set, (2) is directly referred to by a reference in the young generation (in which no remaining objects have been found unreachable), or (3) is referred to by any reference in an object thereby evacuated. All of the objects thus evacuated are placed in cars in the youngest train,

25 which was newly created during the collection cycle. Certain of the mechanics involved in the evacuation process are described in more detail in connection with similar evacuation performed, as blocks 112 and 114 indicate, in response to remembered-set entries.

Fig. 9 illustrates how the processing represented by block 114 proceeds. The entries identify heap regions, and, as block 116 indicates, the collector scans the thus-

30 identified heap regions to find references to locations in the collection-set. As blocks 118

and 120 indicate, that entry's processing continues until the collector finds no more such references. Every time the collector does find such a reference, it checks to determine whether, as a result of a previous entry's processing, the referred-to object has already been evacuated. If it has not, the collector evacuates the referred-to object to a (possibly new) car in the train containing the reference, as blocks 122 and 124 indicate.

As Fig. 10 indicates, the evacuation operation includes more than just object relocation, which block 126 represents. Once the object has been moved, the collector places a forwarding pointer in the collection-set location from which it was evacuated, for a purpose that will become apparent presently. Block 128 represents that step. (Actually, there are some cases in which the evacuation is only a "logical" evacuation: the car containing the object is simply re-linked to a different logical place in the collection sequence, but its address does not change. In such cases, forwarding pointers are unnecessary.) Additionally, the reference in response to which the object was evacuated is updated to point to the evacuated object's new location, as block 130 indicates. And, as block 132 indicates, any reference contained in the evacuated object is processed, in an operation that Figs. 11A and 11B (together, "Fig. 11") depict.

For each one of the evacuated object's references, the collector checks to see whether the location that it refers to is in the collection set. As blocks 134 and 136 indicate, the reference processing continues until all references in the evacuated object have been processed. In the meantime, if a reference refers to a collection-set location that contains an object not yet evacuated, the collector evacuates the referred-to object to the train to which the evacuated object containing the reference was evacuated, as blocks 138 and 140 indicate.

If the reference refers to a location in the collection set from which the object has already been evacuated, then the collector uses the forwarding pointer left in that location to update the reference, as block 142 indicates. Before the processing of Fig. 11, the remembered set of the referred-to object's car will have an entry that identifies the evacuated object's old location as one containing a reference to the referred-to object. But the evacuation has placed the reference in a new location, for which the remembered set of the referred-to object's car may not have an entry. So, if that new location is not as far

19

forward as the referred-to object, the collector adds to that remembered set an entry identifying the reference's new region, as blocks 144 and 146 indicate. As the drawings show, the same type of remembered-set update is performed if the object referred to by the evacuated reference is not in the collection set.

5   Now, some train-algorithm implementations postpone processing of the references contained in evacuated collection-set objects until after all directly reachable collection-set objects have been evacuated. In the implementation that Fig. 10 illustrates, though, the processing of a given evacuated object's references occurs before the next object is evacuated. So Fig. 11's blocks 134 and 148 indicate that the Fig. 11 operation is
10   completed when all of the references contained in the evacuated object have been processed. This completes Fig. 10's object-evacuation operation, which Fig. 9's block 124 represents.

        As Fig. 9 indicates, each collection-set object referred to by a reference in a remembered-set-entry-identified location is thus evacuated if it has not been already. If the
15   object has already been evacuated from the referred-to location, the reference to that location is updated to point to the location to which the object has been evacuated. If the remembered set associated with the car containing the evacuated object's new location does not include an entry for the reference's location, it is updated to do so if the car containing the reference is younger than the car containing the evacuated object.
20   Block 150 represents updating the reference and, if necessary, the remembered set.

        As Fig. 8's blocks 112 and 114 indicate, this processing of collection-set remembered sets is performed initially only for entries that do not refer to locations in the oldest train. Those that do are processed only after all others have been, as blocks 152 and 154 indicate.

25   When this process has been completed, the collection set's memory space can be reclaimed, as block 164 indicates, since no remaining object is referred to from outside the collection set: any remaining collection-set object is unreachable. The collector then relinquishes control to the mutator.

Figs. 12A-12J illustrate results of using the train algorithm. Fig. 12A represents a generation in which objects have been allocated in nine car sections. The oldest train has four cars, numbered 1.1 through 1.4. Car 1.1 has two objects, A and B. There is a reference to object B in the root set (which, as was explained above, includes live objects in the other generations). Object A is referred to by object L, which is in the third train's sole car section. In the generation's remembered sets 170, a reference in object L has therefore been recorded against car 1.1.

Processing always starts with the oldest train's earliest-added car, so the garbage collector refers to car 1.1's remembered set and finds that there is a reference from object L into the car being processed. It accordingly evacuates object A to the train that object L occupies. The object being evacuated is often placed in one of the selected train's existing cars, but we will assume for present purposes that there is not enough room. So the garbage collector evacuates object A into a new car section and updates appropriate data structures to identify it as the next car in the third train. Fig. 12B depicts the result: a new car has been added to the third train, and object A is placed in it.

Fig. 12B also shows that object B has been evacuated to a new car outside the first train. This is because object B has an external reference, which, like the reference to object A, is a reference from outside the first train, and one goal of the processing is to form trains into which there are no further references. Note that, to maintain a reference to the same object, object L's reference to object A has had to be rewritten, and so have object B's reference to object A and the inter-generational pointer to object B. In the illustrated example, the garbage collector begins a new train for the car into which object B is evacuated, but this is not a necessary requirement of the train algorithm. That algorithm requires only that externally referenced objects be evacuated to a newer train.

Since car 1.1 no longer contains live objects, it can be reclaimed, as Fig. 12B also indicates. Also note that the remembered set for car 2.1 now includes the address of a reference in object A, whereas it did not before. As was stated before, remembered sets in the illustrated embodiment include only references from cars further back in the order than the one with which the remembered set is associated. The reason for this is that any

other cars will already be reclaimed by the time the car associated with that remembered set is processed, so there is no reason to keep track of references from them.

The next step is to process the next car, the one whose index is 1.2. Conventionally, this would not occur until some collection cycle after the one during which car 1.1 is collected. For the sake of simplicity we will assume that the mutator has not changed any references into the generation in the interim.

Fig. 12B depicts car 1.2 as containing only a single object, object C, and that car's remembered set contains the address of an inter-car reference from object F. The garbage collector follows that reference to object C. Since this identifies object C as possibly reachable, the garbage collector evacuates it from car set 1.2, which is to be reclaimed. Specifically, the garbage collector removes object C to a new car section, section 1.5, which is linked to the train to which the referring object F's car belongs. Of course, object F's reference needs to be updated to object C's new location. Fig. 12C depicts the evacuation's result.

Fig. 12C also indicates that car set 1.2 has been reclaimed, and car 1.3 is next to be processed. The only address in car 1.3's remembered set is that of a reference in object G. Inspection of that reference reveals that it refers to object F. Object F may therefore be reachable, so it must be evacuated before car section 1.3 is reclaimed. On the other hand, there are no references to objects D and E, so they are clearly garbage. Fig. 12D depicts the result of reclaiming car 1.3's space after evacuating possibly reachable object F.

In the state that Fig. 12D depicts, car 1.4 is next to be processed, and its remembered set contains the addresses of references in objects K and C. Inspection of object K's reference reveals that it refers to object H, so object H must be evacuated. Inspection of the other remembered-set entry, the reference in object C, reveals that it refers to object G, so that object is evacuated, too. As Fig. 12E illustrates, object H must be added to the second train, to which its referring object K belongs. In this case there is room enough in car 2.2, which its referring object K occupies, so evacuation of object H does not require that object K's reference to object H be added to car 2.2's remembered set. Object G is evacuated to a new car in the same train, since that train is where refer-

ring object C resides. And the address of the reference in object G to object C is added to car 1.5's remembered set.

Fig. 12E shows that this processing has eliminated all references into the first train, and it is an important part of the train algorithm to test for this condition. That is, even though there are references into both of the train's cars, those cars' contents can be recognized as all garbage because there are no references into the train from outside of it. So all of the first train's cars are reclaimed.

The collector accordingly processes car 2.1 during the next collection cycle, and that car's remembered set indicates that there are two references outside the car that refer to objects within it. Those references are in object K, which is in the same train, and object A, which is not. Inspection of those references reveals that they refer to objects I and J, which are evacuated.

The result, depicted in Fig. 12F, is that the remembered sets for the cars in the second train reveal no inter-car references, and there are no inter-generational references into it, either. That train's car sections therefore contain only garbage, and their memory space can be reclaimed.

So car 3.1 is processed next. Its sole object, object L, is referred to inter-generationally as well as by a reference in the fourth train's object M. As Fig. 12G shows, object L is therefore evacuated to the fourth train. And the address of the reference in object L to object A is placed in the remembered set associated with car 3.2, in which object A resides.

The next car to be processed is car 3.2, whose remembered set includes the addresses of references into it from objects B and L. Inspection of the reference from object B reveals that it refers to object A, which must therefore be evacuated to the fifth train before car 3.2 can be reclaimed. Also, we assume that object A cannot fit in car section 5.1, so a new car 5.2 is added to that train, as Fig. 12H shows, and object A is placed in its car section. All referred-to objects in the third train having been evacuated, that (single-car) train can be reclaimed in its entirety.

A further observation needs to be made before we leave Fig. 12G. Car 3.2's re-membered set additionally lists a reference in object L, so the garbage collector inspects that reference and finds that it points to the location previously occupied by object A. This brings up a feature of copying-collection techniques such as the typical train-algorithm implementation. When the garbage collector evacuates an object from a car section, it marks the location as having been evacuated and leaves the address of the ob-ject's new location. So, when the garbage collector traces the reference from object L, it finds that object A has been removed, and it accordingly copies the new location into object L as the new value of its reference to object A.

In the state that Fig. 12H illustrates, car 4.1 is the next to be processed. Inspection of the fourth train's remembered sets reveals no inter-train references into it, but the inter-generational scan (possibly performed with the aid of Fig. 6's card tables) reveals inter-generational references into car 4.2. So the fourth train cannot be reclaimed yet. The garbage collector accordingly evacuates car 4.1's referred-to objects in the normal man-ner, with the result that Fig. 12I depicts.

In that state, the next car to be processed has only inter-generational references into it. So, although its referred-to objects must therefore be evacuated from the train, they cannot be placed into trains that contain references to them. Conventionally, such objects are evacuated to a train at the end of the train sequence. In the illustrated imple-mentation, a new train is formed for this purpose, so the result of car 4.2's processing is the state that Fig. 12J depicts.

Processing continues in this same fashion. Of course, subsequent collection cy-cles will not in general proceed, as in the illustrated cycles, without any reference changes by the mutator and without any addition of further objects. But reflection re-veals that the general approach just described still applies when such mutations occur.

In the Train algorithm, as discussed herein, trains and cars are ordered so that typically older cars are collected before younger cars. This ordering helps alleviate a major burden, imposed by the Train algorithm, of the maintenance of per car remembered sets tracking references between objects in different cars. Because of the ordering, there

need only be tracking of references from objects in younger cars to ones in older cars. Nonetheless, maintaining these remembered sets is costly.

It is an objective of the present invention to perform the scanning for reference locations for insertion into remembered sets on data structures whose relevance is restricted
5 to the collector concurrently with the application as much as is advantageous.

SUMMARY OF THE INVENTION

In view of the foregoing background discussion, the present invention provides a garbage collection method and apparatus for inserting references into remembered sets
10 concurrent with operating with application programs. A card table is used to track dirtied (a term of art indicating a recent change) memory sections or cards. A load instruction is used to locate the dirtied regions of the card table and atomic instructions are used to update those locations. The atomic instructions as known in the art are used to update the card table locations in a manner that preserves the integrity of the information during the
15 updating. Compare-and-Swap, CAS, is one instruction, and Load-Locked/Store-Conditionally, LL/SC, is another pair of instructions that can be used for these purposes.

The modified cards are scanned and corresponding remembered sets updated. An atomic operation is used to ensure that changes were not made by the application during summarization. If changes were concurrently made, the card table contents are preserved
20 indicating the modifications for later handling. In some embodiments, an attempt is made to preserve as much of the summarization information as possible without re-doing the summarization.

Since an application is operating in this memory area, the collector moves on to summarize other areas of memory. If no modification were made the collection contin-
25 ues as normally arranged or scheduled.

BRIEF DESCRIPTION OF THE DRAWINGS

The invention description below refers to the accompanying drawings, of which:

Fig. 1, discussed above, is a block diagram of a computer system in which the present invention's teachings can be practiced;

Fig. 2 as, discussed above, is a block diagram that illustrates a compiler's basic functions;

Fig. 3, discussed above, is a block diagram that illustrates a more-complicated compiler/interpreter organization;

Fig. 4, discussed above, is a diagram that illustrates a basic garbage-collection mechanism;

Fig. 5, discussed above, is a similar diagram illustrating that garbage-collection approach's relocation operation;

Fig. 6, discussed above, is a diagram that illustrates a garbage-collected heap's organization into generations;

Fig. 7, discussed above, is a diagram that illustrates a generation organization employed for the train algorithm;

Figs. 8A and 8B, discussed above, together constitute a flow chart that illustrates a garbage-collection interval that includes old-generation collection;

Fig. 9, discussed above, is a flow chart that illustrates in more detail the remembered-set processing included in Fig. 8A;

Fig. 10, discussed above, is a block diagram that illustrates in more detail the referred-to-object evacuation that Fig. 9 includes;

Figs. 11A and 11B, discussed above, together form a flow chart that illustrates in more detail the Fig. 10 flow chart's step of processing evacuated objects' references;

Figs. 12A-12J, discussed above, are diagrams that illustrate a collection scenario that can result from using the train algorithm;

Figs. 13A and 13B together constitute a flow chart that illustrates a collection interval, as Figs. 8A and 8B do, but illustrates optimizations that Figs. 8A and 8B do not include;

Fig. 14 is a diagram that illustrates example data structures that can be employed to manage cars and trains in accordance with the train algorithm;

Fig. 15 is a diagram that illustrates data structures employed in managing different-sized car sections;

Fig. 16 is a block diagram illustrating memory cards and a corresponding card table tracking card changes,

Fig. 17 is a diagram of bytes with particular meanings,

Fig. 18 is a block flow chart showing handling of the card table information and the corresponding cards,

Fig. 19 is a block flow chart expanding the handling of a card table and the corresponding cards, and

Fig. 20 is a block flow chart further expanding the handling of the card table and corresponding cards.

DETAILED DESCRIPTION OF AN ILLUSTRATIVE EMBODIMENT

The illustrated embodiment employs a way of implementing the train algorithm that is in general terms similar to the way described above. But, whereas it was tacitly assumed above that, as is conventional, only a single car section would be collected in any given collection interval, the embodiment now to be discussed may collect more than a single car during a collection interval. Figs. 13A and 13B (together, "Fig. 13") therefore depict a collection operation that is similar to the one that Fig. 8 depicts, but Fig. 13 reflects the possibility of multiple-car collection sets and depicts certain optimizations that some of the invention's embodiments may employ.

Blocks 172, 176, and 178 represent operations that correspond to those that Fig. 8's blocks 102, 106, and 108 do, and dashed line 174 represents the passage of control from the mutator to the collector, as Fig. 8's dashed line 104 does. For the sake of efficiency, though, the collection operation of Fig. 13 includes a step represented by block 180. In this step, the collector reads the remembered set of each car in the collec-

tion set to determine the location of each reference into the collection set from a car out-
side of it, it places the address of each reference thereby found into a scratch-pad list as-
sociated with the train that contains that reference, and it places the scratch-pad lists in
reverse-train order. As blocks 182 and 184 indicate, it then processes the entries in all
scratch-pad lists but the one associated with the oldest train.

Before the collector processes references in that train's scratch-pad list, the col-
lector evacuates any objects referred to from outside the old generation, as block 186 in-
dicates. To identify such objects, the collector scans the root set. In some generational
collectors, it may also have to scan other generations for references into the collection set.
For the sake of example, though, we have assumed the particularly common scheme in
which a generation's collection in a given interval is always preceded by complete col-
lection of every (in this case, only one) younger generation in the same interval. If, in
addition, the collector's promotion policy is to promote all surviving younger-generation
objects into older generations, it is necessary only to scan older generations, of which
there are none in the example; i.e., some embodiments may not require that the young
generation be scanned in the block-186 operation.

For those that do, though, the scanning may actually involve inspecting each sur-
viving object in the young generation, or the collector may expedite the process by using
card-table entries. Regardless of which approach it uses, the collector immediately
evacuates into another train any collection-set object to which it thereby finds an external
reference. The typical policy is to place the evacuated object into the youngest such
train. As before, the collector does not attempt to evacuate an object that has already
been evacuated, and, when it does evacuate an object to a train, it evacuates to the same
train each collection-set object to which a reference the thus-evacuated object refers. In
any case, the collector updates the reference to the evacuated object.

When the inter-generational references into the generation have thus been proc-
essed, the garbage collector determines whether there are any references into the oldest
train from outside that train. If not, the entire train can be reclaimed, as blocks 188
and 190 indicate.

As block 192 indicates, the collector interval typically ends when a train has thus been collected. If the oldest train cannot be collected in this manner, though, the collector proceeds to evacuate any collection-set objects referred to by references whose locations the oldest train's scratch-pad list includes, as blocks 194 and 196 indicate. It removes them to younger cars in the oldest train, again updating references, avoiding duplicate evacuations, and evacuating any collection-set objects to which the evacuated objects refer. When this process has been completed, the collection set can be reclaimed, as block 198 indicates, since no remaining object is referred to from outside the collection set: any remaining collection-set object is unreachable. The collector then relinquishes control to the mutator.

We now turn to a problem presented by popular objects. Fig. 12F shows that there are two references to object L after the second train is collected. So references in both of the referring objects need to be updated when object L is evacuated. If entry duplication is to be avoided, adding remembered-set entries is burdensome. Still, the burden in not too great in that example, since only two referring objects are involved. But some types of applications routinely generate objects to which there are large numbers of references. Evacuating a single one of these objects requires considerable reference updating, so it can be quite costly.

One way of dealing with this problem is to place popular objects in their own cars. To understand how this can be done, consider Fig. 14's exemplary data structures, which represent the type of information a collector may maintain in support of the train algorithm. To emphasize trains' ordered nature, Fig. 14 depicts such a structure 244 as including pointers 245 and 246 to the previous and next trains, although train order could obviously be maintained without such a mechanism. Cars are ordered within trains, too, and it may be a convenient to assign numbers for this purpose explicitly and keep the next number to be assigned in the train-associated structure, as field 247 suggests. In any event, some way of associating cars with trains is necessary, and the drawing represents this by fields 248 and 249 that point to structures containing data for the train's first and last cars.

Fig. 14 depicts one such structure 250 as including pointers 251, 252, and 253 to structures that contain information concerning the train to which the car belongs, the previous car in the train, and the next car in the train. Further pointers 254 and 255 point to the locations in the heap at which the associated car section begins and ends, whereas pointer 256 points to the place at which the next object can be added to the car section.

As will be explained in more detail presently, there is a standard car-section size that is used for all cars that contain more than one object, and that size is great enough to contain a relatively large number of average-sized objects. But some objects can be too big for the standard size, so a car section may consist of more than one of the standard-size memory sections. Structure 250 therefore includes a field 257 that indicates how many standard-size memory sections there are in the car section that the structure manages.

On the other hand, that structure may in the illustrated embodiment be associated not with a single car section but rather with a standard-car-section-sized memory section that contains more than one (special-size) car section. When an organization of this type is used, structures like structure 250 may include a field 258 that indicates whether the heap space associated with the structure is used (1) normally, as a car section that can contain multiple objects, or (2) specially, as a region in which objects are stored one to a car in a manner that will now be explained by reference to the additional structures that Fig. 15 illustrates.

To deal specially with popular objects, the garbage collector may keep track of the number of references there are to each object in the generation being collected. Now, the memory space 260 allocated to an object typically begins with a header 262 that contains various housekeeping information, such as an identifier of the class to which the object belongs. One way to keep track of an object's popularity is for the header to include a reference-count field 264 right in the object's header. That field's default value is zero, which is its value at the beginning of the remembered-set processing in a collection cycle in which the object belongs to the collection set. As the garbage collector processes the collection-set cars' remembered sets, it increments the object's reference-count field each time it finds a reference to that object, and it tests the resultant value to determine

whether the count exceeds a predetermined popular-object threshold. If the count does exceed the threshold, the collector removes the object to a "popular side yard" if it has not done so already.

Specifically, the collector consults a table 266, which points to linked lists of normal-car-section-sized regions intended to contain popular objects. Preferably, the normal car-section size is considerably larger than the 30 to 60 bytes that has been shown by studies to be an average object size in typical programs. Under such circumstances, it would be a significant waste of space to allocate a whole normal-sized car section to an individual object. For reasons that will become apparent below, collectors that follow the teachings of the present invention tend to place popular objects into their own, single-object car sections. So the normal-car-section-sized regions to which table 266 points are to be treated as specially divided into car sections whose sizes are more appropriate to individual-object storage.

To this end, table 266 includes a list of pointers to linked lists of structures associated with respective regions of that type. Each list is associated with a different object-size range. For example, consider the linked list pointed to by table 266's section pointer 268. Pointer 268 is associated with a linked list of normal-car-sized regions organized into n-card car sections. Structure 267 is associated with one such region and includes fields 270 and 272 that point to the previous and next structure in a linked list of such structures associated with respective regions of n-card car sections. Car-section region 269, with which structure 267 is associated, is divided into n-card car sections such as section 274, which contains object 260.

More specifically, the garbage collector determines the size of the newly popular object by, for instance, consulting the class structure to which one of its header entries points. It then determines the smallest popular-car-section size that can contain the object. Having thus identified the appropriate size, it follows table 266's pointer associated with that size to the list of structures associated with regions so divided. It follows the list to the first structure associated with a region that has constituent car sections left.

Let us suppose that the first such structure is structure 267. In that case, the collector finds the next free car section by following pointer 276 to a car data structure 278.

This data structure is similar to Fig. 14's structure 250, but in the illustrated embodiment it is located in the garbage-collected heap, at the end of the car section with which it is associated. In a structure-278 field similar to structure 250's field 279, the collector places the next car number of the train to which the object is to be assigned, and it places

5   the train's number in a field corresponding to structure 250's field 251. The collector also stores the object at the start of the popular-object car section in which structure 278 is located. In short, the collector is adding a new car to the object's train, but the associated car section is a smaller-than-usual car section, sized to contain the newly popular object efficiently.

10 The aspect of the illustrated embodiment's data-structure organization that Figs. 14 and 15 depict provides for special-size car sections without detracting from rapid identification of the normal-sized car to which a given object belongs. Conventionally, all car sections have been the same size, because doing so facilitates rapid car identification. Typically, for example, the most-significant bits of the difference between the gen-

15 eration's base address and an object's address are used as an offset into a car-metadata table, which contains pointers to car structures associated with the (necessarily uniform-size) memory sections associated with those most-significant bits. Figs. 14 and 15's organization permits this general approach to be used while providing at the same time for special-sized car sections. The car-metadata table can be used as before to contain point-

20 ers to structures associated with memory sections whose uniform size is dictated by the number of address bits used as an index into that table.

In the illustrated embodiment, though, the structures pointed to by the metadata-table pointers contain fields exemplified by fields 258 of Fig. 14's structure 250 and Fig. 15's structure 267. These fields indicate whether the structure manages only a single

25 car section, as structure 250 does. If so, the structure thereby found is the car structure for that object. Otherwise, the collector infers from the object's address and the structure's section_size field 284 the location of the car structure, such as structure 278, that manages the object's special-size car section, and it reads the object's car number from that structure. This inference is readily drawn if every such car structure is positioned at

30 the same offset from one of its respective car section's boundaries. In the illustrated ex-

ample, for instance, every such car section's car structure is placed at the end of the car section, so its train and car-number fields are known to be located at predetermined offsets from the end of the car section.

Applications employ "write barriers," well known in the art, to notify the collector of changes made to objects in the garbage-collected heap. Techniques typically include the use of card tables or the use of some form of logging structure, such as sequential store buffers. For purposes of simplification, the following discussion concentrates on use of card tables. However, the approach of summarizing modified reference locations concurrently with the application is applicable to these other approaches to implementing write-barriers.

Referring back, Fig. 6 shows card tables arrange for each of three generations with a single-byte card table entry associated with each card (a section of the generation memory). As discussed above, the contents of the card table may be a binary indication placed by the mutator that a write operation has modified a reference location in the corresponding card. In other embodiments, the card table may contain offsets that indicate write-modified reference locations, and will thereby track locations that have been modified by the application.

Fig. 16 shows a card table 302 with eight bytes corresponding to a region of memory consisting of eight cards, one byte for each card. One issue for modern processors that operate on word lengths of four or eight bytes is the efficiency for handling a granularity of a single byte. Another issue is synchronizing and handshaking between the collector and the application.

In order to facilitate, usually minimizing, the synchronization and handshaking that might be necessary between the collector and the application, atomic operations like "compare-and-swap" (CAS) are used. "Atomic" is a well known term in the art, referring to the fact that no other store can be performed between the load and store elements of an "atomic" instruction. With respect to the atomic CAS, once begun, no other processor can access the memory location specified until the CAS operation (if any) has completed and is potentially visible to all other processors in the system. The preferred CAS operates on multiple bytes, typically four to eight.

As known in the art, CAS may take several forms, one form is: CAS(addr, old_value, new_value). This instruction will compare the contents of addr with old_value, and if they agree CAS is said to have succeeded, and the contents of "addr" is replaced with new_value. If the contents of addr and old_value do not agree, the CAS fails indicating some other process has modified the contents of the location. In either case of success or failure, the CAS operation returns the contents of the location.

The operation of the CAS (compare and swap) instruction above may be more easily understood by the following short noted code:

```
CAS(addr, old_value, new_value) {
        val := *addr
        If (val == old_value) {
                *addr := new_value;
        }
        return val;
}
```

As mentioned above, CAS instructions operate on full word lengths of 4 or 8 bytes (32 or 64 bits) and not directly at the byte level. However, by setting up appropriate before and after values for the CAS word, modification to objects can be tracked at the byte level in the card table. In this preferred embodiment, Fig. 16, eight 512-byte memory cards 303 and the corresponding eight card table bytes 302 are discussed. Fig. 17 shows an 8 byte sequence 310 from the card table where the first two bytes 304 have been "dirtied" or modified to indicate changes in the associated cards by the mutator. In this instance consider all the bits in each dirtied byte are zeros, and undirtied bytes contain all ones indicating empty. For embodiments outlined in Figures 18 and 19, we need to distinguish entries in the card table indicating dirtied references that we are currently scanning from those entries that may be dirtied after we begin summarizing cards associated with a particular sequence of entries. To this end, we also reserve a SCAN value (254), if needed, that will indicate those dirtied entries we are currently scanning. In one preferred embodiment, the other values (1 to 254 when not reserving a SCAN value, or 1

to 253 when such a SCAN value is reserved) in the card table byte are used to record off-sets of references in the memory card to younger generations. In one example, if there is one reference to an object in a younger generation, the byte values other than 0 and 255 (expressed in decimal) indicate an offset from the start of the memory card of a reference

5   to the younger generation object. In the present invention, the atomic CAS operation is used on the 8 byte card table word 302 without locks thereby allowing concurrent modification by an application. The approach is to use a CAS operation on the card table word to isolate dirtied cards, and then to scan and reconcile the references in the associated card. The CAS is then used to ensure that the application made no concurrent

10   changes. But if there were concurrent changes, the card table word is left alone or an attempt is made to reconcile the newly scanned summary information with the modified state, and the collector moves on to other cards in order to avoid working on the same card table entries where the application is operating.

Figs. 18, 19, and 20 are alternative flow charts illustrating the present invention.

15   Each assumes that the collector performs a CAS 310 on the card table word 206 and finds the two bytes 304 containing zeros indicating the two corresponding cards have been dirtied 312. Since the processing of dirtied card table entries revert the dirtied entries to empty, in this example consider all the other bytes 306 to be empty and filled with ones.

Fig. 18 starts at the beginning of the card table 320. If the card table has been

20   fully processed 322, the system has completed a pass summarizing potentially modified locations. But if the card table has not yet been completely processed 324, the next eight   ATG
bytes are loaded 326 into a location V for processing. The bytes are checked for zeros or SCAN
values
328 indicating dirtied memory cards. If any of the checked bytes indicate the presence of dirtied memory cards, a copy of the eight bytes is made that is identical to the previously

25   read eight bytes except any that are zero (indicating a dirty card) are changed to be the SCAN value. A CAS is performed to change the eight entries in the card table from the previously read values to the new values. If successful, the new entries have replaced the old ones in the card table 331 and the entries with SCAN values may now be scanned. Having succeeded the new sequence including SCAN values becomes the current value

ATG   30   of V. If the CAS fails 333, additional entries in the eight-byte sequence of the card table

must have been marked as dirty by the application, and so, to avoid summarizing refer-

ences the application is currently modifying, control is returned to inquiring if the card

table is exhausted. The corresponding dirtied memory cards are scanned 330, remem-

bered sets are updated and references from younger generations are summarized. When

the dirtied memory cards have been processed the contents of the word V' are updated

into location V'. The locations V' and V'' are processed with a CAS instruction 332 and

control returns to see if the card table has been completed or exhausted. It does not mat-

ter if the CAS succeeds or fails, because any dirtied locations that occurred while the

memory cards were being processed are retained in V for a future collection. If the CAS

succeeds then all of the memory cards whose card table entries had been marked dirty

prior to the scanning of the cards have had their entries updated to reflect the results of

the scanning. In either case the collector continues on as before. Fig. 19 is identical to

the operations described for Fig. 18 up to the point of forming a V'' 330. A CAS is per-

formed 332 from location V' to V'' (in this terminology V' is the old value and V'' the new

value) where, if the CAS succeeds 334, then no modifications were made during the

processing of the card table and updating references. However, if the CAS fails 336,

since modification were made concurrently, then for each newly dirtied byte in V' the cor-

responding byte in V'' is dirtied by placing a zero in that byte. In this manner the newly

dirtied bytes are recorded while maintaining the just-performed summarization for card

table entries that have not just been dirtied; the newly dirtied entries will eventually be

processed during a later collection interval.

Fig. 20 is yet another preferred embodiment. In this flow chart, items 320, 322,

324, 326 and 328 operate as in Figs. 18 and 19. However, if there are dirtied bytes in V

340, V' is formed by replacing the dirtied bytes in V with empty values (ones) and pre-

serving the other byte values 342. So bytes having zeros in V are replaced with ones in

the corresponding bytes in V.' A CAS operation 344 from V to V' is performed and if it

fails 346 further processing of these cards will wait until later and operation returns to

interrogating the other entries from the card table 323.

If the CAS succeeds 348, the dirtied memory cards as determined from V are

scanned 350 and remembered sets updated and younger generation references summa-

rized with the result being placed in a new eight byte V." If the contents of V' are identical to V" 352 then no further processing is needed on these memory cards. If they are not identical 354 a CAS is performed 356 from V' to V" returning V.' If this CAS succeeds the operation returns to interrogating the card table, but if it fails for each dirty byte in V'

5 the corresponding byte in V" is dirtied 358, and operation returns to item 353 where V' is checked to see if it matches V."

ARe

Other atomic operations may be used above instead of the CAS operation. Some modern processors, for example, provide a pair of instructions, load-locked (LL) and store-conditional (SC) that serve the same purpose as, and instead of, the CAS operation.

10 On these processors, LL/SC are suitable for implementing the invention. Similarly, more limited atomic instructions (such as the SPARCV9 architecture's load-store-unsigned-byte (LDSTUB) instruction that atomically reads a byte and changes its contents to all ones) may be used for selectively setting indicators to empty if the empty state is represented by all ones.

15 Concurrent scanning may be performed either by a dedicated set of threads performing collection work concurrently with the application or by the application's threads, themselves, at points in their operation such as when allocating memory or when a certain number of writes have occurred in a particular thread. Concurrent scanning may be initiated when a particular amount of memory has been allocated in one or more genera-

20 tions, when a certain amount of time has elapsed, or as part of a concurrent phase of collection of a particular generation, such as one based on the Train algorithm, wherein the collection of the generation requires that modified reference locations be examined.

What is claimed is: